Dictionary-Based Instruction Compression and the TMS320C6000 Architecture

Andrew Aarestad
Joseph Thomas
Kyaw Min

1. Introduction

As solid-state memory technology develops and becomes cheaper, design parameters for devices that use it can change as well. For years, one of the primary goals of computer engineering, especially in the embedded systems field, was the reduction of the memory footprint requirements of software. Now that memory is becoming increasingly larger and more affordable, the design focus is shifting away from footprint reduction to reduction of system power consumption. Power consumption has always been a concern for computer engineers, but only recently has it moved to the forefront of engineering design metrics. New methods of data storage are being developed that not only reduce memory footprint requirements but also reduce memory bus traffic, a major contributor to the overall system power consumption [1]. Instruction compression is one of the techniques being applied to reduce power consumption.

In this paper, we present several schemes that we have developed for use in the Texas Instruments TMS320C6x VLIW architecture. We explore the various performance tradeoffs of modifications to the original architecture, adding compression to the scheme in several places.

2. Design Parameters

Energy Savings

The primary design goal of the compression system is to reduce the energy requirements of the system. This is achieved by increasing the cache hit rate. Fewer cache misses means the CPU has to go off-chip to satisfy its instruction requests less. However, we do not provide an analysis of the gains in terms of power savings since the power requirements of the memory system varies by implementation. Instead, we provide an analysis of the hit rate gains which, given information about the power consumption of the memory bus, could be converted to data about the power consumption.

Transparency

One of the design goals for this project is transparency to the CPU core. This means that one of our requirements is that modifications to the core are unnecessary and unwanted. Our modifications include positioning a decompression unit between different levels of instruction cache. This does mean modification of the chip since the cache is on-chip, but does not require modification of the core. The idea is to allow the core to behave exactly as if there were no modifications. This is an important feature when designing for processors with proprietary cores such as the one we are using.

Cache Bus Widths

The original system as defined in TI's documentation calls for L1 and L2 caches line sizes of 512 and 1024 bits, respectively [2]. The TMS is a VLIW processor, which means that it processes a fetch packet of up to eight instructions per clock cycle. Each instruction is 32 bits, so the cache lines for the L1 and L2 caches represent 2 and 4 fetch packets, respectively. To implement the compression schemes we have proposed, redesign of the cache busses would be required since fewer bits would need to be transferred to satisfy the same requests.

Chip Cost

We have attempted to design our compression schemes in such a way that the cost of the chip does not increase. Since we have allocated cache-speed memory for use in decompression, we have 'borrowed' the memory from the L2 cache. Thus, in our analysis we have reduced the size of the L2 cache to keep the total memory requirements of the chip constant.

3. Compression Schemes

Both of these modifications of the TI TMS320C6x architecture compress machine instructions by placing an instruction decompression unit on the chip (Fig 1). Instructions are compressed before they are placed in main memory from 32 to 16 bits using a dictionary compression algorithm similar to those seen in [3] and [4]. The compressed version of the program and a decompression dictionary are loaded into the DSP's memory. The decompression unit has a small amount of local memory that is used to hold the dictionary. When the CPU requests a 256-bit fetch packet located at a certain address, the decompression unit intercepts the request, translates the address into the compressed address space, fetches the compressed instructions, and decompresses them before returning them to the CPU. The difference between the two schemes we propose here is the location of the decompression unit in the cache system.

The following descriptions of what happens during an instruction fetch illustrate our changes to the original system and give a side-by-side comparison of the actions of both our schemes.

Scheme 1

1. Address is generated in CPU core

2. Address is sent to L1 cache. As in the original configuration, the L1 cache contains uncompressed fetch packets. If the requested fetch packet is not located in the L1 cache, the cache updates.

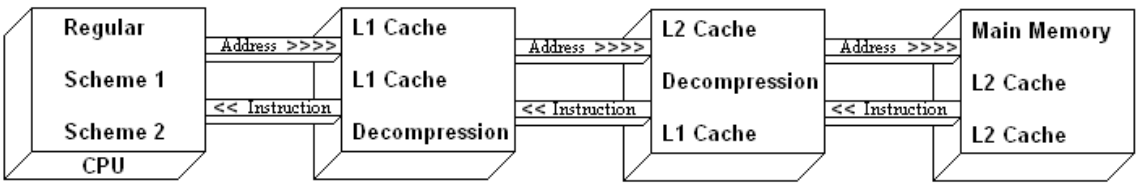

Figure 1 - Shows the placement of the decompression unit for the schemes.

3. Fetch packet is returned if there is a L1 cache hit, otherwise the address is sent to the decompression unit.
4. Address is translated into compressed address space. Since the com pressed instructions are half the size of the original, the address of a compressed instruction will be half of the original request.
5. Address is sent to the L2 cache.
6. The L2 cache is searched for the compressed address.
7. Compressed cache line is returned to decompression unit if there is a L2 hit, otherwise the requested address is sent to the DMA unit.
8. Requested instructions are fetched from main memory.
9. L2 cache line is received from main memory, L2 cache updated.
10. Compressed cache line is sent to decompression unit.
11. Cache line is decompressed.
12. Decompressed cache line is sent to the L1 cache.
13. L1 cache is updated.
14. Decompressed fetch packet is sent to CPU.

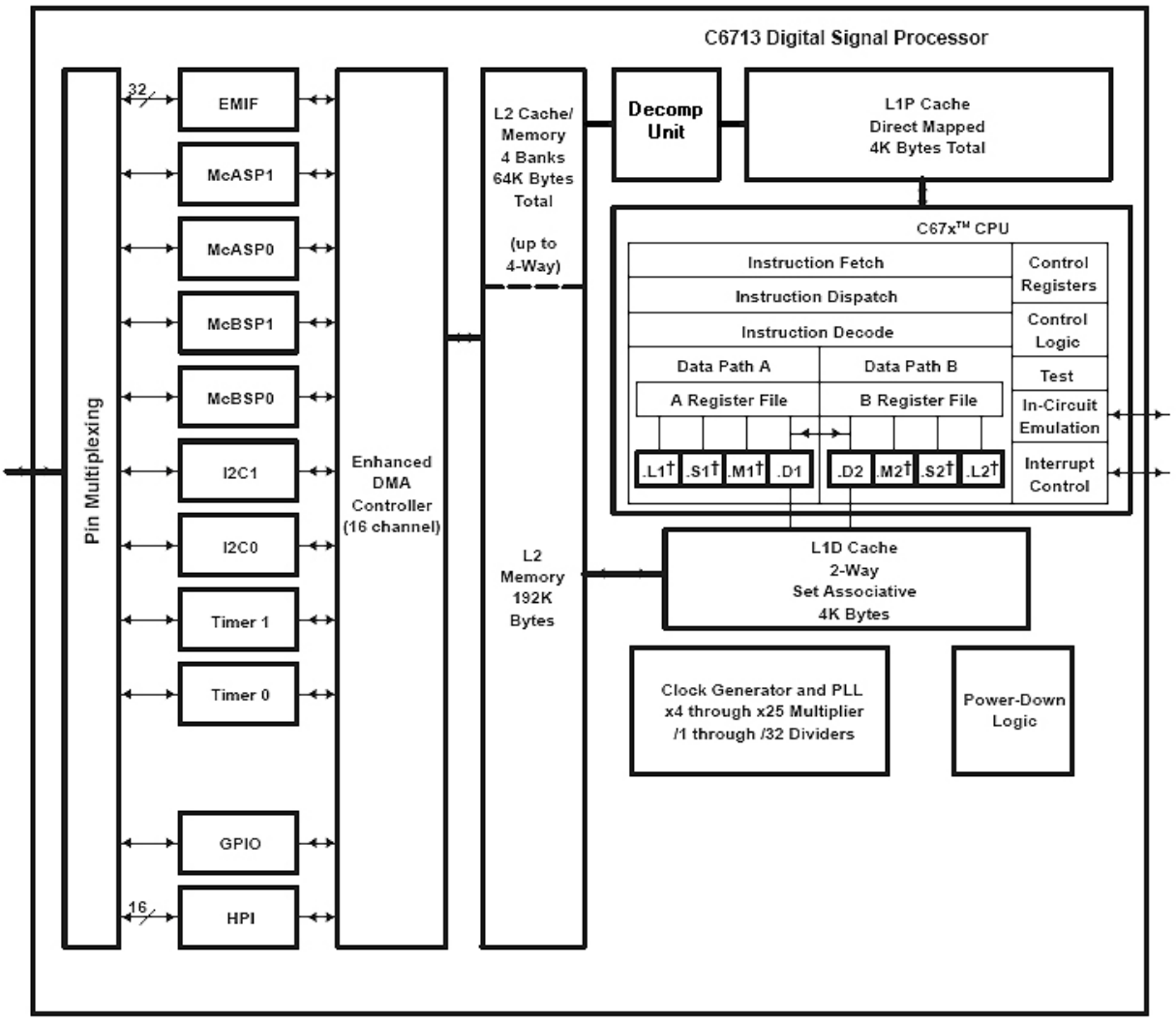

Figure 2 - The TMS320C6713 architecture [5] with decompression unit inserted for scheme 1.

Scheme 2

1. Address is generated in CPU core
2. Address is sent to decompression unit
3. Address is translated into compressed address space
4. Compressed address is sent to L1 cache
5. L1 is searched, if it misses, the address is sent to the L2 cache
6. L2 is searched, if it misses, it is updated from main memory
7. Instructions are fetched from main memory
8. Instructions are received, L2 frame is updated
9. Compressed cache line is sent to L1
10. L1 cache line is updated
11. Compressed fetch packet is sent to decompression unit
12. Fetch packet is decompressed
13. Decompressed fetch packet is sent to the CPU

4. Performance Analysis

We have evaluated the performance of our compression schemes using the following metrics:

- Memory Footprint Reduction
- Change in Cache Hit Rate
- Instruction Access Time Penalty

Memory Footprint

Both schemes use static-length 16-bit codewords. This gives us an even 50% reduction of the program size, but the file must also contain the decompression dictionary. In smaller programs, this may actually result in code expansion. See Fig. 2 for data relating to memory footprint performance of our schemes.

Cache Hit Rate

The primary goal of our compression was to increase the effective cache hit rate. By compressing the caches, we are able to store more fetch packets in cache for the same cache size. With more chances for a cache hit, there will be fewer cache misses.

Access Time Penalty

The tradeoff that comes with the cache hit rate increase is a penalty in instruction access time. A compressed fetch packet is half as long as an uncompressed one. Because the memory controller must bring the instructions of the fetch packet into the chip in serial, only half as many of these fetches would occur per fetch packet, reducing the L2 miss penalty significantly. However, the penalty is increased by the delay caused by decompression. For a basis for the ratio of the access times for different levels of cache, we consulted [6].

5. Simulation Environment

To analyze the performance of our schemes, we simulated several DSP-specific applications on Vinodh Cuppu's c6xsim cycle-accurate simulator [7]. We then ran the instruction trace generated through a cache performance simulator that we developed, TMSCache. We keep track of how many of the instructions issued by the trace are not found in the caches, i.e. cache misses. Many formulas for cache performance involve the number of misses per one thousand instructions [6]. To normalize our data, we keep track of the number of misses during the second 1000 instructions.

During a cycle of TMSCache, the simulator first reads in the next instruction address from the trace. The caches are then searched for the address. If a miss occurs, the cache contents and performance statistics are updated. To update the caches, we implement the descriptions of the system architecture given by TI in [5]. The L1 cache is direct mapped, and we do not use an eviction policy, which would further increase cache performance. The L2 cache is 4-way set associative.

Level	1	2	3
Name	registers	cache	main memory
Typical size	< 1 KB	< 16 MB	< 16 GB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM
Access time (ns)	0.25–0.5	0.5–25	80–250
Bandwidth (MB/sec)	20,000–100,000	5000–10,000	1000–5000
Managed by	compiler	hardware	operating system
Backed by	cache	main memory	disk

Figure 3 - Cache Level Access Times

To estimate a basis for the access time penalty incurred by these compression schemes, we keep track of each instruction's access time and find the average. To do this, we increment a counter for each instruction based on the number of steps required in satisfying the request. The amount incremented is based on the delays associated with each part of the memory system. Each instruction's delay is the sum of one or more of the following values:

L1 Hit Time — 1 – This is because the L1 cache is designed to provide the CPU with one fetch packet per cycle. I.e., a L1 hit will be satisfied in one clock cycle.

L2 Hit Time — 20 - This number we based off of a chart in [6] - see Fig. 3. This is a comparison of the different levels of cache and their respective average access times. We assume that our L2 cache will perform similarly to these averages.

Memory Time — 400 - We also based this number from the averages in [6]. Because the memory controller must bring the instructions of the fetch packet onto the chip in serial, only half as many of these fetches would occur per fetch packet. Thus, when the L2 cache is compressed we use only 200 for the fetch time.

Decompression — Same as the L2 hit time. This is because we allocate L2 speed memory for use in decompression. Also, it is assumed that the dictionary cache is large enough to never have a dictionary miss. This is not an accurate assumption, but we feel that the dictionary cache will miss so infrequently as to not have a significant effect on the delay.

6. Conclusions

The results of our simulations show that reasonable performance gains can be realized by implementing a compression scheme such as the ones we've described. With Scheme 1 we have kept the cost of the chip the same, so the increase in power dissipation produced should be outweighed by the reductions seen in memory buy activity.

Scheme 1 has the same L1 cache performance as the original system since it leaves the L1 cache uncompressed. Gains are noticed in the L2 cache hit rate, see Fig. 3 for data. Since the L2 cache is updated serially, the L2 penalty is reduced by compression. This penalty reduction compensates for the penalty increase caused by decompression and produces an overall decrease in average instruction access time. See Fig. 4 for data about the average access times.

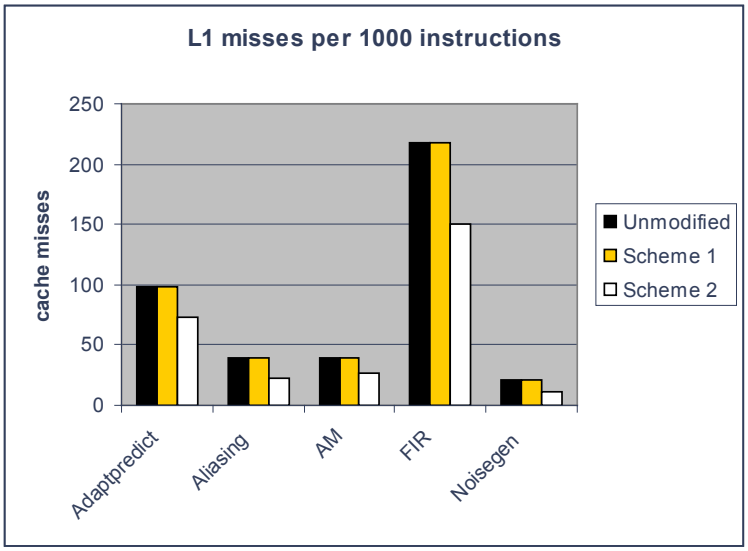

Figure 3 - L1 cache misses

Scheme 2 had better cache miss rates than Scheme 1 or the original, but does so at the cost of access time. Since it must decompress every instruction issued, the decompression penalty adds up fast. As is illustrated in Fig. 3, the average access time is increased by about a factor of three. For a non-time-critical system in which energy savings is paramount, a system such as this could be desirable, but for most systems this is too large of a penalty.

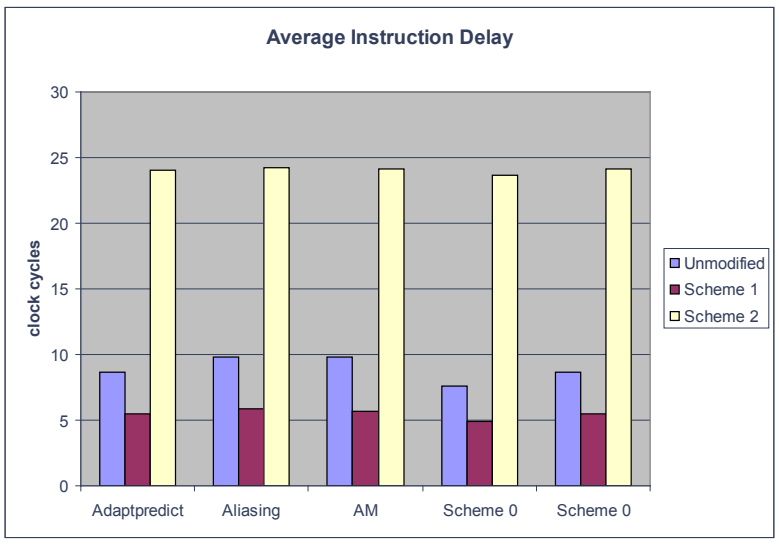

Figure 4 - Average instruction delay for our test programs

In conclusion, our data shows that implementing a compression technique such as the ones we illustrated here can result in both energy savings and performance gains without increasing the cost of the chip.

References

[1] Luca Benini, Alberto Macii, and Enrico Macii, " Minimizing Memory Access Energy in Embedded Systems by Selective Instruction Compression," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 10, No. 5, pp. 521-530, Oct 2002.

[2] TMS320C6713 Floating-Point Digital Signal Processor: Texas-Instruments, 2005.

[3] Luca Benini, Francesco Menichelli, and Mauro Olivieri, "A Class of Code Compression Schemes for Reducing Power Consumption in Embedded Microprocessor Systems," IEEE Transactions on Computers, vol. 53, No. 4, pp. 467-482. Apr. 2004.

[4] Montserrat Ros, and Peter Sutton. "Code Compression Based on Operand-Factorization for VLIW Processors," IEEE Proceedings of the Data Compression Conference, pp. 1-5, Apr 14, 2004.

[5] TMS320C6000 CPU and Instruction Set Reference Guide: Texas-Instruments.

[6] J. L. Hennessy and D.A. Patterson, Computer Architecture – A Quantitative Approach, 3rd ed. San Mateo, CA: Morgan Kaufmann, 1996.

[7] Vinodh Cuppu, Cycle Accurate Simulator for TMS320C62x, 8 way VLIW DSP Processor: University of Maryland, College Park, 1999.